Heap Space Analysis of Hume Programs

Ludwig-Maximilians Universität, München

Jointly with:

Kevin Hammond, Steffen Jost
University of St Andrews

Greg Michaelson, Robert Pointon
Heriot-Watt University Edinburgh

Jocelyn Serot, Norman Scaife
University of Clermont-Ferrand,

Christian Ferdinand, Reinhold Heckmann
AbsInt GmbH Saarbrücken

http://www.embounded.org/

Overview

- Formal cost models of resource consumption
- Foundations of the Heap Analysis
- Generic resource inference for Hume
- Applications
- Summary

Formal Foundations of Hume

Embedded systems crave for formal methods to

- *guarantee* resource bounds;
- provide *compositional* models of resource consumption;
- develop *provably correct* compilers

Our current focus is on semantics and cost models for Hume.

Semantics and cost model are the formal basis for

- *analysis* of resource consumption, to prove soundness of analysis w.r.t. semantics and
- *certification* of low-level code, to embed the cost model into a prover and use it for certificate validation

Steffen to add some basics here, or in a separate talk beforehand

A generic resource analysis for Hume

Goal: Development of a generic resource analysis for Hume

Produce a *generic, re-usable* resource analysis, that can be instantiated for many different kinds of resources (heap, stack, time, . . .).

Characteristics of the analysis:

- Based on a port of the existing heap space analysis for ARTHUR to Hume's intermediate language.
- Abstracts over the costs of individual language constructs.
- Integrated into the Hume compiler (*in progress*), which requires
  - separate phase for Hindley-Milner type inference
  - simplifying transformations on the Hume code (let-normal-form)

Future work: make use of size information (where available) in the resource analysis and use inference results for optimisations and such.

Format of inferred information

Size (resource) information is attached as *weights* to the types.

$$ins :: 1, int \rightarrow list(\ldots\langle 1\rangle) \rightarrow list(\ldots\langle 0\rangle), 0$$

says that the call `ins x xs` requires 1 heap-cell plus 1 heap cells for each `Cons` cell of the list `xs` i.e. $n+1$ heap cells, where n is the length of `xs`.

Sized types are produced by a separate module in the Hume compiler: $ins :: int \rightarrow list^{n} \rightarrow list^{n+1}$

Main advantages of this analysis:

- Efficient (linear programming as computational core)
- Modular (info attached to types)
- Generic (only constants have to be replaced for other resources)

Principles of the amortised costs analysis

Reminder, style of the judgement:

$$\Gamma \vdash^{m}_{m'} e : \tau \mid \Phi$$

where Γ is a variable environment, e is an expression, τ is an enriched type, and C is a set of constraints.

Resource consumption is encoded as *enriched types* for Hume expressions.

Example rule from the type system:

$$\frac{\Gamma_{(i-1)} \vdash^{n_{(i-1)}}_{n_i} e_i : \tau_i \mid \Phi_i \quad (\text{for } i = 1, \ldots, k) \qquad \Sigma(\mathit{fid}) = (\tau_k, \ldots, \tau_1) \xrightarrow[q]{p} \tau}{\Gamma_0 \dot{\cup} \ldots \dot{\cup} \Gamma_k \vdash^{n_0}_{n'} \mathit{fid}\ e_k \cdots e_1 : \tau \mid \bigcup_{1 \le i \le k} \Phi_i \cup \{n_k \ge p, n_k - p + q \ge n'\}}$$

(HEAP COST CALL)

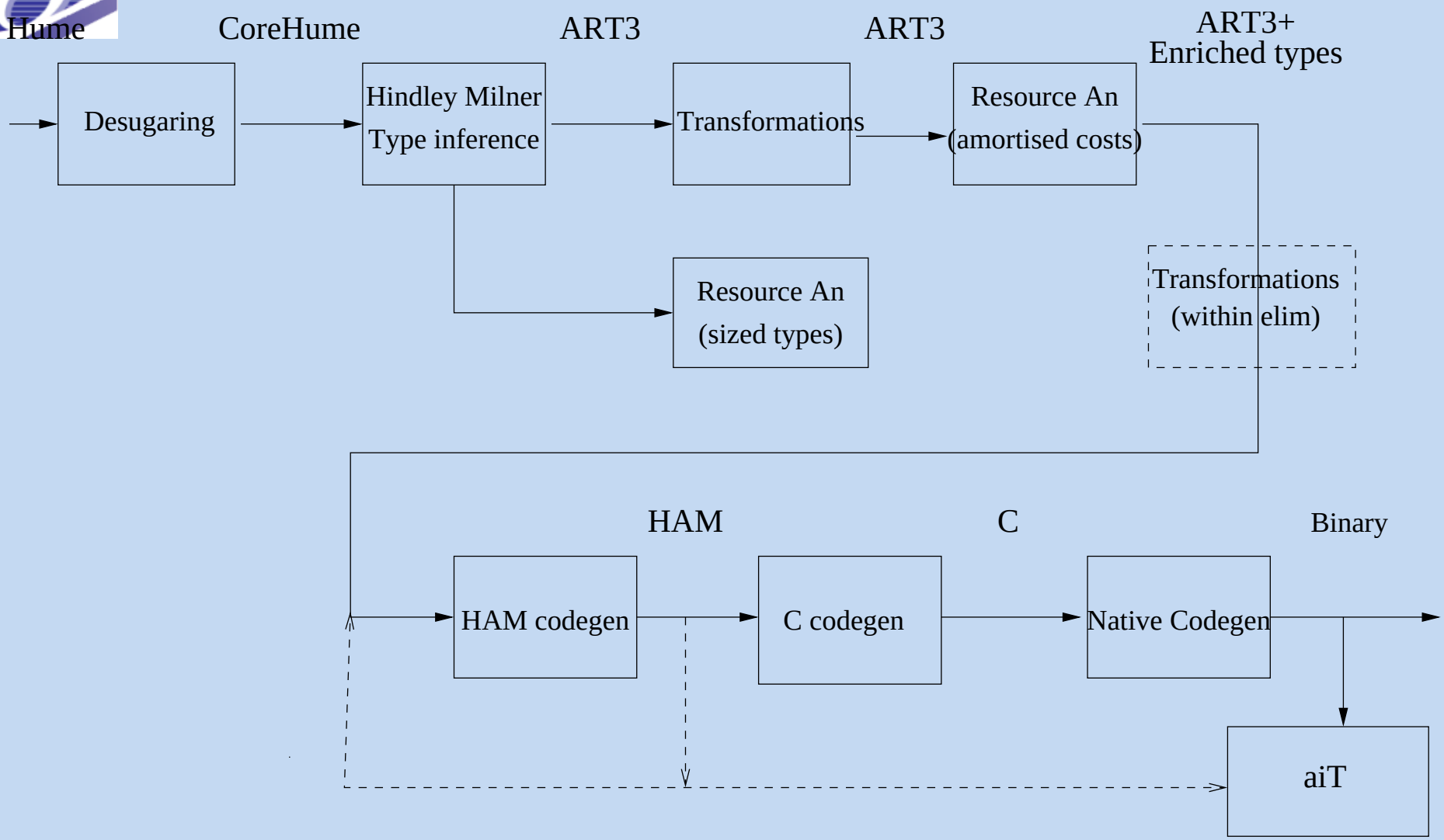
Current state of implementation

Resource being analysed: Heap-space

Output format: enriched type, ie. annotations to algebraic data-types and functions

Input language: expression-level Hume with the following restrictions:

- no nested patterns
- case-expressions rather than top-level pattern matching
- . . .


Hume
CoreHume
ART3
ART3
ART3+
Enriched types
Desugaring
Hindley Milner
Type inference
Transformations
Resource An
(amortised costs)
Resource An
(sized types)
Transformations
(within elim)
HAM
C
Binary
HAM codegen
C codegen
Native Codegen
aiT

Examples of Heap Analysis on Hume Code

Hume

```
insert x xs =
   case xs of
      [] -> x:[]
    | (y:ys) -> if x < y then x : y : ys
                         else y : insert x ys;
```

rewritten to

```
insert (Pair x xs) =
  case xs of
     Nil ->  Cons x Nil
   | (Cons y ys) -> if x < y then (Cons x (Cons y ys))
                             else (Cons y (insert (Pair(x,ys))));
```

ARTHUR code for insert

```
let rec insert :: iil_pair -> ilist = \ (iil::iil_pair) ->
 case iil of
 | Pair(e,l1) -> let x = case l1 of
                  | Nil -> let ?t0 = Nil
                           in  Cons(e,?t0)
                  | Cons(h,t) -> let v= (e <= h)
                                 in
                                 if b
                                   then
                                     let ?t1 = Cons(h,t)
                                     in Cons(e,?t1)
                                   else
                                     let ?t2 = Pair(e,t) in
                                     let ?t3 = insert ?t2 in
                                     Cons(h,?t3)
                           esac
   in x
 esac
in insert
```

The Result

Resource information is printed as an enriched type:

```
ARTHUR Type of main expression:
  iil_pair[INT,ilist[0|INT,#,1],0]-1/0->ilist[0|INT,#,0]
```

Basic usage of the inference

To analyse a suitable expression-level Hume program `foo.hume` type

```
art3me foo.hume
```

This will generate output including the ARTHUR3 type of the expression statement.

You can read the heap space consumption off this type.

The intermediate file `foo.art3` is the input to the size inference, proper.

Stages:

- `.hume`$\rightarrow$`.art3`: `phamc -r foo.hume`
- `.art3`$\rightarrow$ARTHUR3 type: `art3`

These are locate in a directory that is printed if you type `ls -l 'which art3me'`.

Practicals

Analyse the following expression-level Hume programs:

- ra1.hume
- fo1.hume
- t3.hume
- recmult2.hume
- recmult3.hume

Modify the following programs: